Project 1

Ethan Febinger, Ryan Mao
CS 440: Intro to AI

February 19, 2021

1 Preliminaries

1. *Write an algorithm for generating a maze with a given dimension and obstacle density p.*

   See function `gen_maze` in `Maze.py`.
   The function takes in two parameters: the first parameter is the desired dimension of the maze, and the second parameter is the desired obstacle density. The maze is represented as a list of lists (a 2D list). The resulting maze will have a 0 to represent an open square, and a 1 to represent a square with an obstacle.

2. *Write a DFS algorithm that takes a maze and two locations within it, and determines whether one is reachable from the other. Why is DFS a better choice than BFS here? For as large adimension as your system can handle, generate a plot of 'obstacle density p' vs 'probability that S can be reached from G'.*

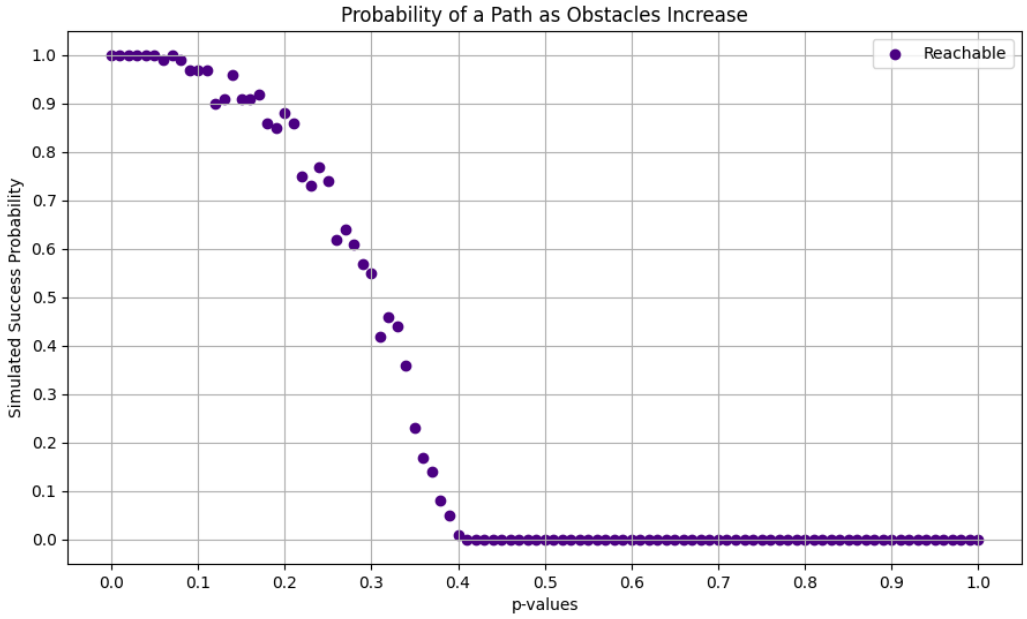
   See function `reachable` in `Maze.py`.
   The function uses DFS to determine whether or not there exists a path between the start square and the goal square. The function returns a boolean: `True` if there exists a path, and similarly, `False` if you cannot get to the goal from the start. The function does not return any paths.

   DFS is a better choice than BFS here since we only wish to determine whether or not the maze is possible to traverse given a start and goal square. Therefore, there is no need to find an optimal path, only a valid path. Due to the nature of the DFS algorithm, it explores as deep as it can before considering branching sideways. In contrast, BFS branches as much as it can before stepping one square deeper. In other words, DFS will go as far as it can until there are no more nodes for it to visit, so it explores nodes farther from the start square (and closer to the goal square), first. If there exists no path, DFS and BFS will take the same amount of time as all squares will be explored regardless.

For the graph of 'obstacle density p' vs 'probability that S can be reached from G', we define probability as:

$$\frac{Number\_of\_Successes}{Total\_Sample\_Space}$$

Below is a scatter plot generated by the function `generate_reachable_plot` in `Graphs.py`.



The data for this graph was gathered by generating 100 mazes at each p value and counting how many of those mazes had a valid path. The mazes were of size 100. The p values ranged from 0 to 1 (inclusive on both sides), with a step of 0.01.

The graph resembles some sort of logistics graph since probability is bounded between 0 and 1. The graph is also always decreasing. The slope decreases slowly at first, then begins to decrease faster, but finally flattens out towards the end.

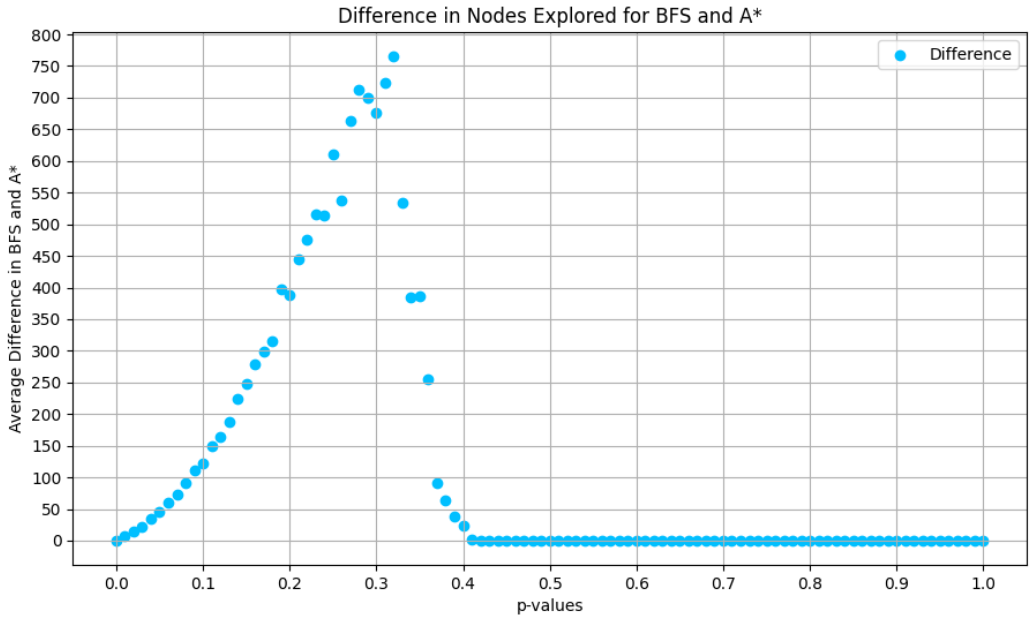
We see that the probability of a valid path occurring is practically 0 at a p value of 0.4 and larger.

3. *Write BFS and A* algorithms (using the euclidean distance metric) that take a maze and determine the shortest path from S to G if one exists. For as large a dimension as your system can handle, generate a plot of the average 'number of nodes explored by BFS - number of nodes explored by A*' vs 'obstacle density p'. If there is no path from S to G, what should this difference be?*

   See function `BFS` in `Maze.py`. See function `AStar` in `Maze.py`.

   For the graph of 'number of nodes explored by BFS - number of nodes explored by A*' vs 'obstacle density p', we generate a maze using a given dimension and p value, and proceed to run both our BFS and A* algorithms. Both functions return not only a `True` or `False` value to represent the existance of a path, but also an optimal path, and the number of squares checked to find said optimal path. Taking the BFS squares explored value, and subtracting the A* squares explored value, gives us the difference in squares explored between the two. Note that this value is always greater or equal to 0 since A* will always explore less than or equal to BFS (In fact, whenever there does NOT exist a valid path, both functions will explore the same amount of squares: i.e. all possible squares reachable from the start square).

   Below is a scatter plot generated by the function `generate_BFS_AStar_plot` in `Graphs.py`.



   The data for this graph was gathered by generating 100 mazes at each p value and summing the difference between the number of squares explored for the two search algorithms. Then we divide the sum by $1,000$ to get the average difference. The mazes were of size 100. The p values ranged from 0 to 1 (inclusive on both sides), with a step of 0.01.

There seems to be a somewhat linear correlation between p value and the difference in A* vs. BFS from 0 to 0.3. Whether the function is actually linear or potentially exponential is unknown. The graph then quickly and drastically falls to 0 as p approaches 0.4. This is understandable, as according to our reachable plot in **Problem 2**, p values close to 0.4 were practically unreachable. Therefore, A* and BFS would have both been unreachable, and more importantly, both explored the same amount of nodes.

4. *What's the largest dimension you can solve using DFS at p= 0.3 in less than a minute? What's the largest dimension you can solve using BFS at p= 0.3 in less than a minute? What's the largest dimension you can solve using A* at p= 0.3 in less than a minute?*

   See function `search_time` in `Maze.py`.
   This function, when given a search algorithm and a maze size, generates mazes until a maze with a valid path is "solved" by the search algorithm (that is a valid/optimal path is found). The return value is the time in seconds that the search algorithm took.

   The largest dimension that DFS can solve at a p value of 0.3 in less than a minute (specifically 59 seconds) is $7,500$. The DFS function uses a replicated version of the maze as a visited matrix, allowing us to access and change visited values in constant time. The downside of this, however, is that we must perform a value by value copy, which is inefficient if we only explore a small amount of squares (e.g. if the starting square is surrounded by obstacles, the DFS function still creates the visited matrix).

   The largest dimension that BFS can solve at p value of 0.3 in less than a minute (specifically 55 seconds) is $4,550$. However, when we have a maze size of $4,600$, the average compute time is 61 seconds. Therefore, the actual maze size to achieve as close to 60 seconds as possible lies within $4,550$ and $4,600$. These results agree with our predictions, in that BFS takes longer than DFS since it computes for an optimal path rather than simply the existance of one.

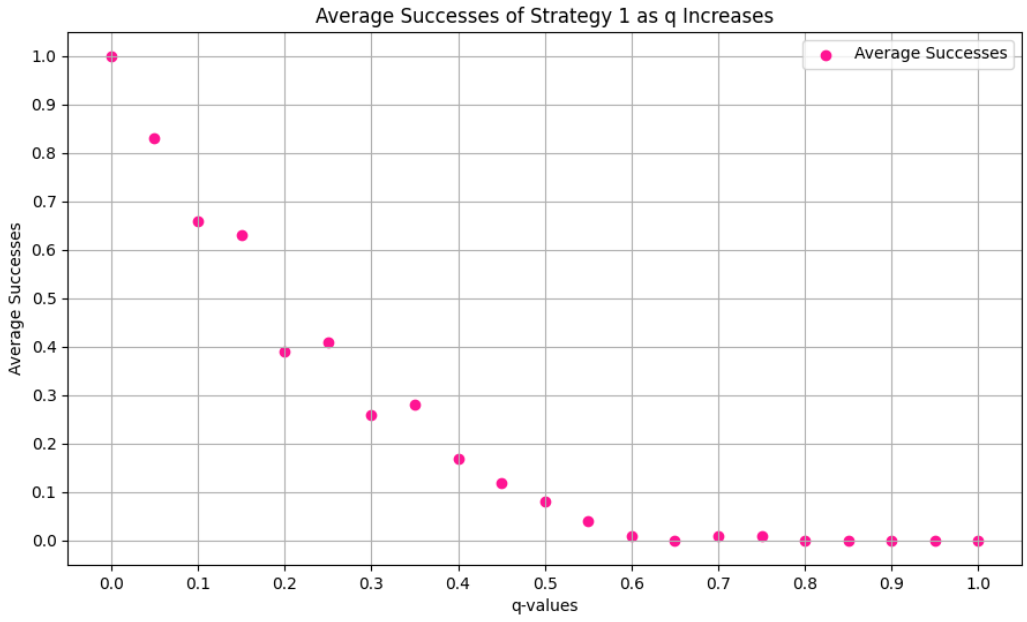
   The largest dimension that A* can solve at p value of 0.3 in less than a minute (or exactly a minute) is $2,750$. We see from these results that the euclidean distance calculation does in fact add a decent amount of computation time.

2 Given Strategies to Solve the Fire Maze

1. *At the start of the maze, wherever the fire is, solve for the shortest path from upper left to lower right, and follow it until the agent exits the maze or burns. This strategy does not modify its initial path as the fire changes.*

   See function `fire_strategy_1` in `Maze.py`.
   Note that this function does not actually discard mazes that are invalid (by invalid, we mean that there does not exist a path from the start to the goal, there does not exist a path from the start to the starting fire square, or both).
   The `generate_strategy_1_plot` in `Graphs.py` is actually the function responsible for ensuring that all graphs provided to `fire_strategy_1` are valid.

   Below is a scatter plot generated by the function `generate_strategy_1_plot` in `Graphs.py`.

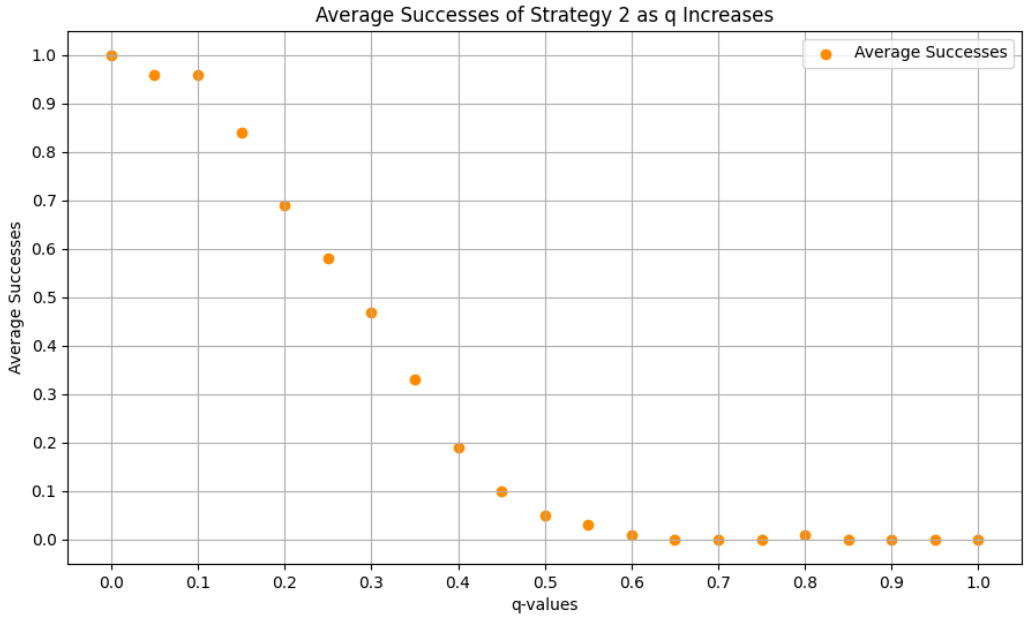


   Since generating the mazes, and performing the fire spreading, is quite computationally expensive, we could not use as many q-values as we would like. The maze size is 100, amd each q value has 10 mazes generated, with each maze then having 10 different fire starting points.

   The graph is always decreasing, and the rate at which it is decreasing is always decreasing. We see that at 0.6 for q, the success rate becomes effectively 0.

2. *At every time step, re-compute the shortest path from the agent's current position to the goal position, based on the current state of the maze and the fire. Follow this new path one time step, then re-compute. This strategy constantly re-adjusts its plan based on the evolution of the fire. If the agent gets trapped with no path to the goal, it dies.*

   See function `fire_strategy_2` in `Maze.py`.
   Similarly to `fire_strategy_1`, this function does not actually discard mazes that are invalid.
   The `generate_strategy_2_plot` in `Graphs.py` is actually the function responsible for ensuring that all graphs provided to `fire_strategy_2` are valid.



   The maze size is 100, amd each q value has 10 mazes generated, with each maze then having 10 different fire starting points.

   In comparison to strategy 1, it seems that strategy 2 faired better in the beginning when q was still close to 0. Both strategy 1 and 2 look to be the same around 0.4 to 0.5, with strategy 1 being marginally higher. After 0.6, both strategies have a practical 0 average successes.

3. *Which strategy performed better?*

   Without a doubt, strategy 2 has higher number of average successes than strategy 1. Clearly, adding the extra computations needed to reevaluate and update the path based on new fire spread information helped the survivability of the agent.